PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 :
G06F 11/20, 11/00, 11/16

A2

(11) International Publication Number: **WO 97/22054**

(43) International Publication Date: 19 June 1997 (19.06.97)

(21) International Application Number: PCT/SE96/01609

(22) International Filing Date: 6 December 1996 (06.12.96)

(30) Priority Data:
9504396-4 8 December 1995 (08.12.95) SE

(71) Applicant *(for all designated States except US)*: TELEFONAKTIEBOLAGET LM ERICSSON (publ) [SE/SE]; S-126 25 Stockholm (SE).

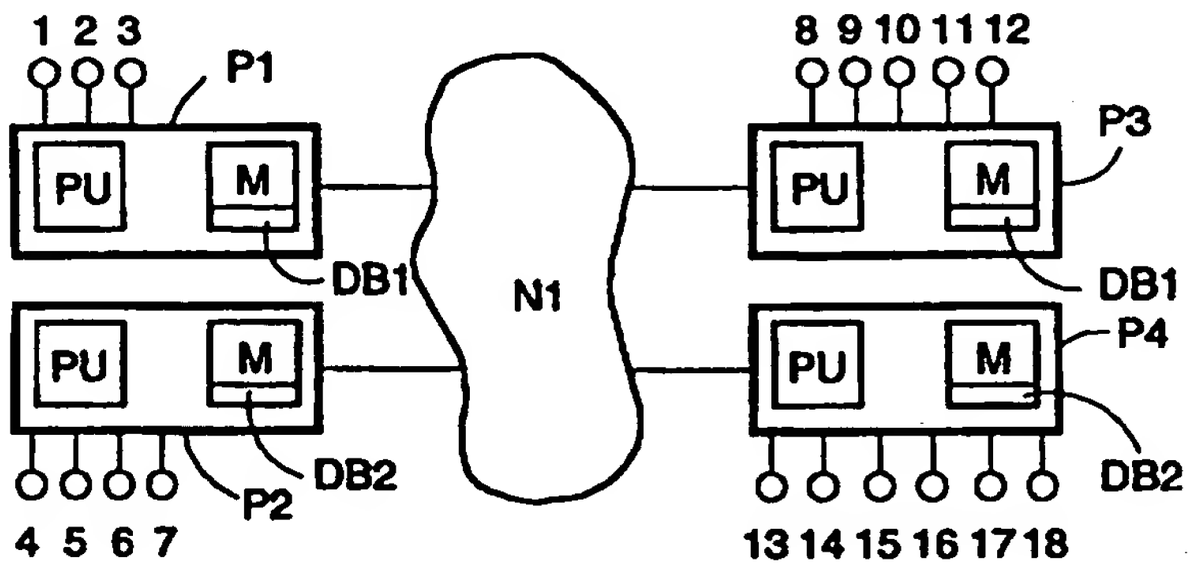
(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: JENSEN, Lars, Ulrik [SE/SE]; Erik Dahlbergsgatan 55, S-115 57 Stockholm (SE).

(74) Agents: BJELLMAN, Lennart et al.; Dr Ludwig Brann Patentbyrå AB, P.O. Box 1344, S-751 43 Uppsala (SE).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, TJ, TM, TR, TT, UA, UG, US, UZ, VN, ARIPO patent (KE, LS, MW, SD, SZ, UG), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: PROCESSOR REDUNDANCY IN A DISTRIBUTED SYSTEM



(57) Abstract

A method of automatically recover from multiple permanent failures of processors in a distributed processor system, in particular a software driven telecommunication system. The method involves the creation of an initial configuration describing each processor and software objects executing thereon, and, for each processor the creation of a catastrophe plan to be followed if the processor has a failure. A catastrophe plan contains information as how to redistribute the software objects executing on the faulty processor to operating processor of the processor system. If a processor goes down its software objects are transferred to operating processors following the catastrophe plan for the faulty processor. A hardware and a software model of the processor system and its software is presented. A software object that has a hardware dependency is handled by the model.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	**GB**	United Kingdom	**MW**	Malawi
AT	Austria	**GE**	Georgia	**MX**	Mexico
AU	Australia	**GN**	Guinea	**NE**	Niger
BB	Barbados	**GR**	Greece	**NL**	Netherlands
BE	Belgium	**HU**	Hungary	**NO**	Norway
BF	Burkina Faso	**IE**	Ireland	**NZ**	New Zealand
BG	Bulgaria	**IT**	Italy	**PL**	Poland
BJ	Benin	**JP**	Japan	**PT**	Portugal
BR	Brazil	**KE**	Kenya	**RO**	Romania
BY	Belarus	**KG**	Kyrgystan	**RU**	Russian Federation
CA	Canada	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CF	Central African Republic			**SE**	Sweden
CG	Congo	**KR**	Republic of Korea	**SG**	Singapore
CH	Switzerland	**KZ**	Kazakhstan	**SI**	Slovenia
CI	Côte d'Ivoire	**LI**	Liechtenstein	**SK**	Slovakia
CM	Cameroon	**LK**	Sri Lanka	**SN**	Senegal
CN	China	**LR**	Liberia	**SZ**	Swaziland
CS	Czechoslovakia	**LT**	Lithuania	**TD**	Chad
CZ	Czech Republic	**LU**	Luxembourg	**TG**	Togo
DE	Germany	**LV**	Latvia	**TJ**	Tajikistan
DK	Denmark	**MC**	Monaco	**TT**	Trinidad and Tobago
EE	Estonia	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**UG**	Uganda
FI	Finland	**ML**	Mali	**US**	United States of America
FR	France	**MN**	Mongolia	**UZ**	Uzbekistan
GA	Gabon	**MR**	Mauritania	**VN**	Viet Nam

TITLE OF INVENTION

Processor redundancy in a distributed system

TECHNICAL FIELD

The present invention relates to a distributed, fault-tolerant reconfigurable processor system in a telecommunication network.

BACKGROUND ART

In public telecommunication systems there are several processors performing many different kinds of tasks, such as monitoring activity on subscriber equipment lines, set up and release of connections, traffic control, system management and taxation. Groups of processors are interconnected by way of a network which is separate from the telecommunication network or forms a part thereof. In modern telecommunication networks there are network elements, such as an exchange, a data base, a processor, that are distributed on several physical elements of the physical network making up the telecommunication network. To an application, such as POTS (Plain Old Telephony Service), GSM (Global System for Mobile communication), VLL (virtual leased lines), BISDN (Broadband Integrated-Services Digital Network), a distributed processor or a distributed data base looks like a single unit. The distributed units are said to bee transparent from the point of view of a distribution.

A main requirement for processor based control systems of public telecommunication systems concerns system availability. With this is meant that the system should be available to serve its users. In for example the AXE-10 telephone system only 2 hours of unavailability of the system during 40 years was allowed. Converted into minutes per year this corresponds to abut 3 min/year. Modern telecommunication systems have much higher availability demands. Nevertheless it is also required of modern telecommunication systems to allow for planned maintenance work, which may have long duration, at long term intervals, for example at intervals in the order of about 1 months.

Generally speaking there is a strive after both improved system availability and longer mean time to repair (MTTR). A long mean

time to repair is desirable allowing for repair on workdays (Monday through Friday) and at work hours (8 PM through 5 PM) and allowing for scheduled maintenance. This leads to a requirement that the telecommunication system should tolerate that one processor goes down and that, before it has been repaired, also a second processor is allowed to go down, and that the system shall tolerate this and be operative. In the worst case, and before the first and second faulty processors have been repaired, the system shall tolerate that also a third, a fourth and additional processors goes down, and that the system is available despite of the faulty processors.

U.S. Patent Serial No. 4 710 926 relates to a fault recovery method for a distributed processing system using spare processors that take over the functions of failed processors. A spare processor acts as a stand-in for one or more active processors. When a spare processor is put into service it will no longer serve as a spare processor for any other processor in the system. During fault-recovery all functions executing on the faulty processor are transferred to the spare processor. The old spare processor's function of being a spare processor is transferred to a second spare processor in the system.

Accordingly the system requires two or more spare processors. When a spare processor is inactive it does not perform any job tasks. When it becomes active it starts processing job tasks - provided it is operative, i.e. is not impaired by any faults. Although not described in said patent it will be necessary to run test programs to verify that the spare processors are operative.

In US Patent Serial No. 4 412 281 a distributed, fault tolerant, reconfigurable signal processing system is described, said system comprising many identical, interconnected sub-system elements sharing the overall processing tasks. The sub-system elements are redundantly interconnected with double busses. Some sub-system elements serve as spare system elements and are ready to take over the tasks of a faulty sub-system element. A spare sub-system element is assigned a self-test task to test all its functions. By rotating spare sub-system elements and active sub-system elements periodically, a distributed operating system ensures that all sub-

system elements execute self-test tasks, thus providing for detecting more subtle faults. Faulty elements of the sub-system are removed from service and replaced by a spare element without system shut down.

Accordingly the processing system uses spare sub-system elements that do not participate in the overall processing tasks. The method used for reconfiguration when a faulty element is detected and replaced by a spare element is one that uses distinct socket addresses for each element in the system. A socket address is assigned a virtual address which replaces the socket address when a faulty condition is detected.

In IEEE Trans. on Parallel and Distributed Systems, vol. 4, no 8, August 1995, p 955-863 an article entitled "Reconfiguration and analysis of fault-tolerant circular butterfly parallel system" by Nian-Feng Tzen describes a network of processors. The network can reconfigure itself when a processor goes down. Spare processors are used to this end. When the system is up an running the spare processors are inactive. When a processor goes down an inactive spare processor becomes active and takes over all of the traffic handled by the faulty processor. As described above the use of spare devices in order to reconfigure a system is a very basic and general alternative to the use of replacement devices. The specific problem addressed by Tzeng is to provide a processor network that after reconfiguration maintains its topology. Provided the topology can be maintained, no addressing reconfiguration is required in case of a processor failure and thus the butterfly architecture task allocation schemes can be maintained. This problem and the solution described in this paper has nothing in common with the problem addressed by the present invention.

In SIGMOND record, Special interest group on management of date, no. 2, 1995, p 11-12 an article entitled "Recovery protocols for shared memory database systems" by L.D. Molesky et al, relates to a database system comprising a number of processors each one of which is provided with a cache memory. When a processor goes down and its cache memory drops out, the dropped out chace memory comprises fragments of transactions relating to data that has been changed in

the chache memory. To recover data from the failure it is generally known to roll back the transactions made on every processor in the system. Moleskey has found, as a side effect of the chache coherency protocol, that data can be recovered by providing a database log which is used to roll back the transactions associated with the failed processor's chache memory only. The transactions performed by the rest of the processors may continue and will not corrupt the data of the database. The problem addressed by Molesky is in no way related to reconfiguration of processor systems. A database log is not like a catastrophe plan.

In Distributed Proceeding - proceedings of the IFIP WW6 10:3 working conference, October 1987, p 133-150 an article entitled "Task redistribution with allocation constraints in a fault-tolerant real-time multiprocessor system" by A-M Deplanche et al addresses the same problem area as the present invention. When a processor failure has been detected the tasks of the failed processor is redistributed to the remaining safe processors. The reconfiguration is done in hard-time by which is meant that the reconfiguration has to take place within a critical time limit, referred to as deadline. Deplance has found a method for accomplishing this. The method is a speedy a heuristic algorithm which describes the reallocation of the faulty processor's tasks before the deadline expires. Deplance thus starts the reallocation process at the time a processor goes down and finishes it before the deadline expires. In the article Deplance indicates that there are methods for computing task allocations off-line, but such methods are complex, require much processor work and provide allocation tables that are very large. This is so because the number of conceivable combinations of tasks and processors is very large even for moderately sized processor systems. Deplance is thus warning for the use of such off-line algorithms. The inventor of the present invention has realized this problem and his contribution to the art is to provide reallocation tables, not for all possible configurations of processors and tasks, but for one configuration only.

SUMMARY OF INVENTION

One object of the invention is to provide a method for automatically recover from multiple permanent failures of processors in a distributed processor system which is used in an application environment of a telecommunication system having high demands on availability while simultaneously allowing system maintenance, planned or unplanned.

Another object of the invention is to utilize available processing resources while allowing for a heterogeneous processing environment due to evolving technology in a system that grows over time and due to particular needs of different parts of an application that runs on the processor system.

Another object of the invention is to provide a method for quickly recovering from multiple permanent failures of processors in a distributed processor system used in a telecom system's environment by providing an initial configuration of all processors and by providing, for each processor in the system, a catastrophe plan to be used in case the corresponding processor goes down. A catastrophe plan is the means by which software objects installed on a faulty processor is distributed to generally several processors in the system thus providing for load sharing among the processors.

A further object of the invention is to have all of the catastrophe calculated and installed in memories associated with the processors so that they are available to the system instantaneously at the time a processor goes down.

Still another object of the invention is to provide new catastrophe plans for the system of operating processors, some of which have installed thereon software objects from a failed processor, so as to prepare the system for a quick recovery should a further processor in the system go down.

Still another object of the invention is to provide a method of the indicated kind which takes back the system to its initial configuration of processors and software objects when the system's

faulty processor or processors after repair or replacement are inserted back into the system.

Another object of the invention is to provide in a catastrophe plan associated with an individual processor an initial redistribution of software objects executing on said individual processor to other non-faulty processors prior to the finishing redistribution of software objects of a faulty processor so as to free up memory for storage of a large software object, which is running on the faulty processor and which in accordance with its catastrophe plan is to be transferred to said predefined processor; the memory which is freed up being the memory associated with said predefined processor.

Another object of the invention is to include in the catastrophe plan redistribution of objects executing on non-faulty processors to other non-faulty processors, so as to free up processor resources, such as memory and CPU capacity, for large software objects which are running on the faulty processor and which in accordance with the faulty processor's catastrophe plan shall be transferred to the processors on which resources have been freed up.

An object of the invention is also to provide a software model that allows software objects to be transferred from a faulty processor to an operating processor by restarting the object on the operating processor. A software model will also allow for killing a software object installed on a processor and for restarting it on a repaired, previously faulty, processor which has been reinserted into the system. This latter objective is predominantly used when the system returns to its initial configuration and there are objects installed on operating processors, which objects should be given back to the repaired processors.

In accordance with the invention there is created a model of the telecommunication system, said model comprising a hardware model of the control processors and the controlled hardware equipment as well as a software model that supports and fits into the hardware model of the telecommunication system.

In accordance with the invention a first algorithm is used to calculate the catastrophe plans for each of the operating processors given either the initial configuration or any one of the actual configurations that will appear after a further processor has gone down.

In accordance with the invention a second algorithm is used that given an actual configuration computes a delta configuration that applied to the actual configuration will give back the initial configuration of the system.

In the software model used for the software objects there are no configuration dependencies encapsulated in the software objects thus allowing a software object to be transferred from one processor to another in any possible processor configuration.

SHORT DESCRIPTION OF THE DRAWINGS

An exemplary embodiment of the invention will be described with reference to the accompanying drawings, wherein

Figure 1 is a block diagram showing a distributed processor system in an initial configuration,

Figure 2 is a block diagram showing the processor system of Figure 1 in an actual configuration after failure of one of the processors,

Figure 3 is a block diagram of the processor system of Figure 1 in a second actual configuration after failure of two processors,

Figure 4 is a flow diagram of the method in accordance with the invention,

Figure 5 is a block diagram of a distributed processor system some of the processors of which are controlling hardware equipment,

Figure 6A is a schematic view of a modularized software object,

Figures 6B-D are block diagrams of three different types of software objects,

Figure 7 is a block diagram illustrating how the hardware and software models in accordance with the invention fit together in one single model of the telecommunication system in accordance with the invention,

Figure 8 is a block diagram showing the hardware model in accordance with the invention, and

Figure 9 is a block diagram of the distributed processor system showing a preparatory redistribution of software objects.

DETAILED DESCRIPTION OF THE INVENTION

In Figure 1 there is shown a number of distributed processors P1, P2, P3 and P4 which communicate over a network N1. The processors form part of a non-shown telecommunication network. The network N1 may form part of said non-shown telecommunication network. Each processor comprises a processor unit PU and memory M. Software objects 1, 2, 3... 18 are installed on the processors; objects 1, 2, 3 on processor P1, objects 4-7 on P2, objects 8-12 on P3 and objects 13-18 on P4.

The software of an application that runs in the telecommunication network comprises software objects (Figs. 6B-D), which are contained in software modules (Fig.6A). The modularized software objects are allocation independent objects that can be transferred freely between the processors. A modularized software object is independent of other modularized software objets. A software object typically comprises a process and persistent data. Persistent data is data that survives a restart of the software object. Software objects can communicate with each other. A task which is required by an application typically involves several software objects on different processors, and is executed by some or all of the processes of these objects. The actual distribution of software

objects on different processors is unknown to the application. Modularized persistent data can be stored in a data base. Following the distributed nature of the processor system the data base is also distributed over several the memories M of several processors, preferably the memories of all of the processors P1, P2, P3 and P4. These data base partitions are labeled DB1, DB2, DB3 and DB4 and comprises a random access memory (RAM). From the view of an application the distributed nature of the data base is transparent. Persistent data must be safely stored. This can be achieved by conventional back up technique, for example by storing the data on a disk. A novel and preferred alternative is, however, to store a mirror copy of each modularized software object in a data base partition of another processor than the one on which said object is installed. In particular the mirror copy of each modularized software object is stored in the database partition on theprocessor given by the catastrophe plan for the processor on which the modularized software object, the original, is executing. In this way copies of the modularized persistent data will be safely stored on another processor if the processor on which the original is installed crashes.

The way the objects 1-18 are distributed among the processors P1-P4 is referred to as the initial configuration of the distributed processor system and is shown in the enclosed Table 1, from which it is evident that for example objects 1, 2 and 3 are installed on processor P1.

The initial configuration must not disappear if any processor goes down. For this reason the initial configuration and the mirror copy thereof is stored as described above. Instead of implementing the initial configuration in the form of a table it can be implemented in so called tuples. As an example tuples (1,1), (1,2), (1,3) would correspond to the information given by the first row of Table 1.

To keep the recovery time of the processor system, and therefore also of the telecommunication system, short should a processor go down there should be a catastrophe plan associated with the faulty processor. Since one cannot forsee the processor that goes down there must be a catastrophe plan associated with each processor. A

catastrophe plan contains directions regarding the processors to which the software objects of the faulty processor should be transferred. One catastrophe plan, shown in Table 2, indicates the processors to which the objects installed on processor P1 should be transferred in case processor P1 goes down. Another catastrophe plan contains information on where the objects installed on processor P2 should be transferred in case processor P2 goes down. Similarly there is a catastrophe plan to follow when processor P3 goes down, said catastrophe plan being indicted in Table 4. Finally there is a catastrophe plan, Table 5, taking care of what to do in case processor P4 goes down.

Processor P1 goes down.

As an example, assume processor P1 goes down. The processor system must quickly recover from this failure and therefore the software objects 1, 2, 3 installed on P1 are transferred to processors P2 and P4 according to the catastrophe plan of P1. In particular software objects 1 and 3 shall be transferred to processor P2 and software object 2 shall be transferred to processor PS4. This is achieved by deleting software objects 1-3 from processor P1 and by creating and starting software objects 1, 3 on processor P2 and by creating and staring software object 2 on processor P4. Of course this will require that there is execution capacity available at processors P2 and P4. Should this not be the case, then the system cannot recover from the processor failure. It is assumed that the necessary processor capacity is available and that the system thus recovers from the failure. The system will now have the configuration shown in Figure 2 and in Table 6. Starting from this configuration, referred to as an actual configuration, new catastrophe plans must be established so that the system can quickly recover if another processor goes down. To this end new catastrophe plans are established, said new catastrophe plans giving directions regarding the processors to which the software objects installed on a faulty processor should be transferred. Since one cannot foresee which one of the three operating processors P2-P4 that will go down, it will be necessary to create catastrophe plans for each one of the operating processors. Table 8 is the new catastrophe plan (CP-P2') for processor P2, Table 9 the new one (CP-P3') for processor P3 and Table 10 the new one (CP-P4')

for the processor P4.

Like the original catastrophe planes the new catastrophe planes and its mirror copies are stored in the above described manner.

It should be noted here that the creation of new catastrophe plans for the actual configuration saves memory as compared to the following theoretically possible scheme: Say the system comprises four processors and that the system is designed to tolerate that two processors goes down. Since one cannot forsee which one that is the first to go down and which one is the second to go down, catastrophe plans in a number corresponding to the number of ways two elements can be selected from a group of four elements must be created stored in the database. Accordingly 24 catastrophe plans must be created and stored. The storing requires much amount of memory and will grow faster than exponentially with the number of processors the system can tolerate to go down.

During the period from the recovery of the system up to the time new catastrophe plans have been worked out and stored in the data base the system is vulnerable. No processor failure must happen during this period. Should a processor go down under this period then the system will not be available.

When a processor P1 after removal and reparation is reinstalled in the system, the system shall revert to the initial configuration. This can be done either by killing all software objects in the actual configuration and by creating and starting all software objects on the processors of the system. In the preferred embodiment of the invention only the objects transferred from the first processor, and which now execute on other processors, are killed at first and are then created and started on processor P1. In order to find out where the objects are a delta configuration table is created by subtracting the initial configuration from the actual configuration, excluding processor P1. By subtracting Table 1 from Table 6 the delta configuration shown in Table 7 is achieved. The row pertaining to the faulty processor P1 is not included in the subtraction. The delta configuration indicates that objects 1 and 3 at processor P2 and object 2 at processor P4 should

be killed at the respective processors. Next, they shall be created and started on the repaired processor P1. After said creation the system is now running like it did in the initial configuration and its recovery time was short.

Processor P1 goes down and then processor P2 goes down.

In the next example to be described it is assumed processor P1 goes down and then processor P2 goes down. Following the previous example it is first supposed that the system is running with the same configuration as shown in Figure 1, that catastrophe plans have been created for each one of the processors P1-P4, that processor P1 crashes, that the software objects installed on processor P1 are transferred to operating processors following the catastrophe plan of Table 2, that the system recovers and is up an running, that new catastrophe plans are created for processors P2, P3 and P4, and that processor P1 is removed and brought to repair. Now it is supposed that processor P2 goes down.

Accordingly, the new catastrophe plan associated with processor P2, i.e. the new catastrophe plan of Table 8 should be followed. According to that plan objects 1, 3 and 4 should be transferred to processors P3 and objects 5-7 should be transferred to processor P4. In the same way as before the software objects on processor P2 are removed and are transferred to the processors P3 and P4. The system will now be up an running and will have a configuration of the kind shown in Figure 3 and Table 11. In accordance with the invention it will now be necessary to work out catastrophe plans for each one of the processors P3 and P4.

If during the period from the time at which the second processor P2 crashed up to the time the new catastrophe plans for processors P3 and P4 have been created and stored in the system, the system is vulnerable. If anyone of P3 or P4 goes down the system will not be available. For the sake of the example it is assumed that this does not happen. Instead the system is up and running with the new configuration shown in Figure 3 and Table 11.

Next the faulty processor P2 is taken away from the system for repair. Next it is assumed that processors P1 and P2 are repaired and are put back into the system. It will now be necessary to kill

objects 1-7 on processors P3 and P4 and to create and start them on their original processors P1 and P2. Using a delta configuration, achieved by subtracting the initial configuration from the actual configuration shown in Table 1, excluding the faulty processors P1 and P2 this time, the objects to kill are identified, in this case objects 1-7. In particular objects 1, 3, 4 on processor P3 and objects 2, 5, 6, 7 on processor P4 should be transferred. The way these objects should be distributed among processors P1 and P2 are given by the initial configuration table. Accordingly, objects 1, 2, 3 are created and started on processor P1 and objects 4-7 are created and started on processor P2. The system has now recovered from the insertion of two repaired processors and is now supposed to be up and running.

In the above examples a processor system comprising four processors has been described. The inventive method is equally well applicable on processor systems that comprise two, three five or more processors. In the last example a processor system tolerating two faulty processors was described. The inventive method is equally well applicable on processor systems that tolerate three or more faulty processors. Each time a processor goes down its software objects are transferred to other operating processors of the system and new catastrophe plans for the operating processors are created. The last example illustrates that a four processor system can operate with 50% of its processors faulty. The application will still run, but it will have a degraded performance. If the processor system is a switch in a local office, telephone traffic will still be running and congestion will start at a low traffic volume. This is a novel and unique feature that is not present in any of the above referenced US patents, and, as far as applicant knows, no one else has achieved before.

A first algorithm is used for creating catastrophe plans from the initial configuration in case a first processor goes down or from the actual configuration in case a further processor goes down is the same. The first algorithm comprises parameters that pertain to the capacity of a processor, parameters that pertain to the size of the memory of a processor, parameters relating to how much processor capacity (machine cycles per process to execute) and

memory the individual objects to be transferred do require, and parameters relating to the quality of service.

A second algorithm is used for returning the system to its initial configuration. This second algorithm has already been described above and has been referred to as a delta configuration.

Various methods can be used to detect a faulty processor, for example the "heart beat" method in accordance with the US Patent Specification 4 710 926 referred to above. A preferred method in a typical telecom network is, however, to monitor the links by which processors are interconnected through the network N1.

From the two examples given above it will be recalled that software objects installed on a faulty processor are transferred to two processors. Of course the faulty processor's objects can also be distributed among three or more processors in the system. As an exceptional case all software objects are transferred to a single processor in case the system comprises two processors that are in working order and one of these crashes.

When the system is up and running, all its processors are operating and have capacity and memory more than necessary for performing their tasks at engineered capacity, i.e. they shall have capacity and memory left for taking over objects from one or more faulty processors and for taking over the application tasks which are being done. This will ensure that all processors are in working order; no test program needs to be run to verify this. Also, the capacity of the system will exceed engineered capacity, which means the system will have "spare" capacity that can be used to take care of additional application tasks; there is no dead "spare" capacity. To the application a crashed processor will imply a degraded system performance only; it will not kill the system.

As an alternative to dimension the system with an active "spare" capacity that can be used for taking care of additional application tasks, the system is designed with no active "spare" capacity and all processors are working with engineered capacity. When a processor goes down the system will work with degraded performance.

In Figure 4 the method steps performed in accordance with the invention are shown in a flow diagram. There is always an initial configuration in which all of the modularized software objects are mapped on individual processors. The initial configuration is created by a system vendor or system operator and is stored in the system. This is indicated in box 20. Next catastrophe planes should be created in accordance with the first algorithm. There should be as many catastrophe planes as there are processors in the system. Further mirror copies of persistent data base objects should be created. This is indicated in box 21. In a preferred embodiment each processor creates its own catastrophe plan, i.e. the catastrophe plan to be used by the system in case it goes down. This will ensure that the work for creating the catastrophe plans will be totally distributed. Next a processor goes down, box 22. The software objects of the faulty processor should be transferred to operating processors using the catastrophe plan for the faulty processor. By transferring objects is contemplated that new copies of the software objects of the crashed processor are created and started on the processors to which they should be transferred in accordance with the catastrophe plan. This is indicated in box 23. Box 23 accordingly represents the recovery of the system from the faulty processor. The system is now up and running and a new configuration, referred to as actual configuration, arises. The actual configuration is also stored in a memory of a distributed processor. While the system is running, new catastrophe planes for the operating processors are created, box 24. Now the system has recovered its ability to withstand a new processor failure. Also mirror copies of the new catastrophe plans are stored in the data base. If a new processor goes down, the process returns to operation 22 as indicated by arrow 25. Next the faulty processor or faulty processors are repaired, box 26, and are inserted into the system, box 26. If two or more processors have crashed, it is assumed they are repaired and that they are inserted back into the system simultaneously. Theoretically it is of course possible to repair faulty processors one by one and insert them into the system one by one but from practical point of view this procedure is roundabout. The last step in the process, box 27, is to take the system back to its initial configuration using the second

algorithm.

In the above described examples it has been assumed that the software objects 1-18 do not control any hardware equipment. Example of hardware equipment controlled by software modules are I/O devices, subscriber line interface devices, subscriber line processors, tone decoders, voice prompting devices, conference equipment. Hardware dependencies of this kind pose restrictions on the software modules. A software module involved in controlling hardware equipment that is connected to one or more processors can not be transferred to an arbitrary processor in the system but must be transferred to a processor that has access to the very same hardware equipment. The catastrophe planes must be created with this in mind.

A telecom system can usually continue to operate despite the loss of some devices, although the services it provides might be somewhat impaired.

Consequently a catastrophe plan must allocate device controlling software objects, to the extent possible, to processors that have access to the controlled devices. When this is not feasible the modularized software object that controls the device must be excluded from the catastrophe plan. An excluded software object is always of the type shown in Figure 6C.

Figure 6B illustrates a software object which contains a function part (execution part) and a persistent data part (persistent part). The software object in Figure 6C contains the function part of the software object shown in Figure 6B and the software object shown in Figure 6D contains the persistent data part of the same software object shown in Figure 6B. The software objects in Figures 6C and 6D together form a pair and have the same key. The key is the logical address to the software object shown in Figure 6B in the data base and the key will therefore also be the logical address to the two software objects of Figure 6C and 6D in the data base. It is the software object of Figure 6C that controls a device. It is in this software object that there is a hardware dependency.

An excluded software object in an actual configuration must be blocked, i.e. other software objects shall not be able to communicate with it. In a preferred embodiment blocking is provided by setting the persistent data part of the software object shown in Figure 6D in blocked state. A blocked state is marked by a setting a flag in the software object. Note that it is not the device controlling software object 6C that is blocked but its persistent companion software object in Figure 6D. By examining the state of a software object before trying to communicate with it, the application can handle blocked devices in an orderly manner.

As an alternative to block flagged software objects in the data base representation thereof, the software object is blocked in the address tables existing in the operating system of the respective processors. The operative system of a processor has address tables to it own software objects. The address tables are used when messages are sent to its object.

In the term quality of service is included the fact that the processors of a processor system can be of two kinds, fault tolerant processors (FTP processors) and non-FTP processors. An FTP processor, which usually comprises double processors, goes on with executing its tasks even if there arises a simple hardware fault in the hardware equipment controlled by the FTP processor. When the fault occurs an alarm will be trigged but the program does not crash. By means not described in the present application, because they do not form part of the present invention, it is possible to take a FTP processor out of the service for repair in a controlled manner, using the catastrophe plans, so that the services the application deliver will not be interrupted. For example, in a telecom system no traffic disturbances will occur; ongoing calls will not be interrupted. Accordingly system recovery in consequence of removal of a FTP processor will not interrupt delivered services. By combining the present invention with the FTP processors described above it is possible to achieve very high system availability as well as high service retainability when hardware failures occurs. In short, the purpose of the quality of service parameter is thus to support FTP processors for software

that requires very high service retainability. An FTP processor will thus mask an internal hardware fault, but the FTP processor must be repaired before new internal hardware faults occur.

In Figure 5 there is shown a processor system similar to Figure 1 where there is a network N1 to which processors P1-P4 have access and can communicated with each other. Although not shown in Figure 5 it is supposed that the software objects 1-18 are distributed on processors P1-P4 in the same way as shown in Figure 1. Further there is a device D1 connected to processor P1. There is also a second network N2 to which processors P2 and P4 have access. Device processors D5 is a device that is used to connect devices D2 and D3 to the network N2. Device processor D5 thus controls devices D2 and D3. Another device processor D6 connects devices D4 and D5 to the network N2 and will thus control these. There are software objects that connect the devices D1, D2, D3...D7 to the processor system. As an example software object 1 is supposed to control device D1. If P1 goes down none of the processors P2-P4 can control device D1. Accordingly object 1 can not be transferred to any of P2-P4. As regards devices D2-D7, however, software objects that control any of these devices must be installed on a processor that can communicate with these devices. Devices D2-D7 can be controlled via the N2 network and accordingly software objects controlling any of these devices can be installed on any of processors P2 and P4. Processors P1 and P3 are out of question. To summarize, a software object that connects hardware to the system must be installed on a processor that has access to the device.

Typical examples of a device of the D1 kind is processor P1 itself. Another example is some hardware device, like a UART-device. If the processor P1 goes down, or if the hardware device controlled by processor P1 goes down, then the software object which represents processor P1 or the software object that represents the faulty hardware cannot be transferred to any of processors P2-P4 since none of these can gain control over the faulty hardware or of the processor P1. Nevertheless, the processor system must tolerate that P1 goes down if the system is to be redundant. When the software object representing processor P1 goes down, the software object that represents processor P1 must be blocked so it cannot be

accessed by any other software objects.

Refer to Figure 1. For each one of the four processors P1-P4 there is an object that describes the processors. In particular object 1 represents processor P1, object 4 represents processor P2, object 8 represents processor P3 and object 13 processor P4. All such hardware dependencies are exactly described by the model of the hardware with installed software as shown in Figure 7. Accordingly the model shows that none of these objects 1, 4, 8 and 13 can be transferred to any other processor in the system. The first algorithm operates on the model of the hardware with installed software and will thus take into account all hardware dependencies when it creates new catastrophe plans. The catastrophe table for processor 1 shown in Table 2 will therefore remain the same with the exception that object 1 disappears. In the catastrophe table for processor 2 (Table 3) object 4 will disappear and in the catastrophe table for processor 3 object 8 will disappear and object 13 will disappear from the catastrophe table associated with processor P4. Accordingly less objects than described previously will remain on the respective processors when a processor goes down.

If for example processor P1 goes down it will be necessary to block object 1 from access from other software objects. This means that no other software objects are allowed to communicate with software object 1.

The model in accordance with the invention will always pretend that the system will operate even if hardware equipment is lost and cannot be controlled by its software objects. If, however, at a higher level of the system it turns out that the system does not operate, not even with impaired services delivered, then the reason why the system does not operate is lack of redundancy and the model cannot change this fact.

In Figure 6A the software model of the modularized software object is shown. The software model comprises a class named configObj which has the operations construct() and destruct(). Construct() and destruct() are used to create and kill respectively a

particular modular software object. The modular software object has been described above with reference to Figures 6B-6D.

Finally the hardware model of the system shown in Figure 1 is described with reference to Figure 8. The hardware model 30 describes the physical devices and their sites in the system. The hardware model 30 comprises a class processor 31 which generates objects that represent processors P1-P4, a class CPPool 32 which generate objects that represent network N1, a class DevX 33 which generates objects that represents network N2 and a class ConfigObj 34 which generate objects which represent the software objects, shown in Figure 6B, of the devices referred to above; device processors inclusive. Class ConfigObj 35 generate software which represent the software objects of the modularized software object shown in Figure 6A but do not form part of the hardware model.

The model shows that processors are connected to N1 and to N2. The model also shows that software which has no devices can be installed on all processors that can connect to N1, while software which controls devices must be installed on processors that can connect to N2. The model will define the constraints for each hardware dependent software object. By connecting the ConfigObj to OrgX not only is the hardware devices included in the hardware model, but at the same time are the software objects that control the devices installed in the model.

Refer to Figure 9. Suppose processor 1 goes down and that the software objects executing thereon shall be redistributed in accordance with its catastrophe plan shown in Table 2. The catastrophe plan is stored distributed on processors P2 and P3 in fragments. In particular catastrophe plan fragment 40 is stored in the memory M of processor P2 and catastrophe plan fragment 41 is stored in memory M of processor P2. In each of the memories M of P1, P2 and P3 software objects and data are stored, as exemplified by the various hatched layers. All memories are not completely filled as exemplified by the non-hatched memory areas. In particular memory M of processor P2 has a free, non-occupied memory area 42 and memory M of processor P3 has a free, non-occupied memory area 43. Likewise the CPU capacity of the different

processors are used to different extents (not necessary in proportion to the memory usage of the respective processor). According to the catastrophe plan of processor P1 software object 1 shall be redistributed to processor P2. The free memory area 42 of processor P2, however, is not large enough for housing object O1. Therefore the catastrophe plan of processor P1 contains an initial redistribution phase in order to make room for software object O1. In the example shown object O4 in the memory of processor P2 is removed and is transferred to the free memory area 43 in processor P3 leaving an enlarged free memory area in processor P2, large enough to house software object O1. To free up memory in processor P2 the objects executing on processor P2 are killed. Following this the objects which in accordance with the catastrophe plan are to execute on processor P2 are created on processor P2. In this manner there will be processor resources (memory as well as CPU capacity)available for executing the new objects.

The object O4 killed on processor P3 must not disappear and is created on another non-faulty processor, for example processor P3.

The above procedure with killing and creating objects on a processor is made for each one of the processors in the system. On processor P3 for example the object O4 is created

For processor P2 the initial redistribution phase and finishing redistribution phase are schematically illustrated by arrows 44 and 45.

Instead of moving just one software object from the memory of an individual processor to the memory of just one other processor many software objects can be transferred from the memory of said individual processor to memories of many other processors in the initial redistribution phase of the catastrophe plan of said individual processor.

Table 1

INITIAL CONFIGURATION	
PROCESSOR-ID	OBJECT-ID
1	1, 2, 3
2	4, 5, 6, 7
3	8, 9, 10, 11, 12
4	13, 14, 15, 16, 17, 18

Table 2

CATASTROPHE PLAN for Processor 1 (CP-P1)	
OBJECT-ID	PROCESSOR-ID
1	2
2	4
3	2

Table 3

CATASTROPHE PLAN for Processor 2 (CP-P2)	
OBJECT-ID	PROCESSOR-ID
4	1
5	3
6	3
7	4

Table 4

CATASTROPHE PLAN for Processor 3 (CP-P3)	
OBJECT-ID	PROCESSOR-ID
8	1
9	1
10	2
11	4
12	4

Table 5

CATASTROPHE PLAN for Processor 4 (CP-P4)	
OBJECT-ID	PROCESSOR-ID
13	2
14	1
15	2
16	1
17	2
18	1

Table 6

ACTUAL CONFIGURATION Table WHEN PROCESSOR 1 IS DOWN	
PROCESSOR-ID	OBJECT-ID
2	1, 3, 4, 5, 6, 7
3	8, 9, 10, 11, 12
4	2, 13, 14, 15, 16, 17, 18

Table 7

Delta configuration for Processor 1	
PROCESSOR-ID	OBJECT-ID
2	1,3
3	-
4	2

Table 8

New CATASTROPHE PLAN for Processor 2 (CP-P2')	
OBJECT-ID	PROCESSOR-ID
1	3
3	3
4	3
5	4
6	4
7	4

Table 9

New CATASTROPHE PLAN for Processor 3 (CP-P3')	
OBJECT-ID	PROCESSOR-ID
8	2
9	2
10	4
11	4
12	4

Table 10

New CATASTROPHE PLAN for Processor 4 (CP-P4')	
OBJECT-ID	PROCESSOR-ID
2	2
13	2
14	2
15	2
16	3
17	3
18	3

Table 11

NEW ACTUAL CONFIGURATION Table WHEN PROCESSOR 2 IS DOWN	
PROCESSOR-ID	OBJECT-ID
3	1, 3, 4, 8, 9, 10, 11, 12
4	2, 5, 6, 7, 13, 14, 15, 16, 17, 18

CLAIMS

1. A method of automatically recover from multiple permanent failures of processors in a distributed processor system of a software driven telecommunication system **characterized by**

- (a) creating an initial configuration describing each processor and software objects executing thereon,
- (b) creating, for each processor, a catastrophe plan to be followed if the processor has a failure, said catastrophe plan containing information as how to redistribute the software objects executing on the faulty processor to operating processor of the processor system,
- (c) redistributing the software objects of a faulty processor to operating processors following the catastrophe plan for the faulty processor,
- (d) running the redistributed software objects on their respective processors thus recovering the processor system from the faulty processor,
- (e) creating, for each of the processors now operating in the system, new catastrophe plans to be followed should any one of the now operating processors go down, whereby the processor system will recover its ability to withstand a new processor failure,
- (f) repeating steps (c)-(e) for each processor that goes down,
- (h) repairing the faulty processors and inserting them into the processor system,
- (i) returning to the initial configuration by distributing back to their respective processors said redistributed software objects.

2. A method of automatically recover from multiple permanent failures in accordance with claim 1 **characterized in that** said initial configuration is created by mapping software objects executing on an individual processor on said individual processor and repeating said mapping for each one of the processors of the processor system.

3. A method of automatically recover from multiple permanent failures in accordance with claim 2 **characterized in that** said step of redistribution comprises creating and starting the software objects to be redistributed on the processors indicated by the catastrophe plan of the faulty processor.

4. A method of automatically recover from multiple permanent failures in accordance with claim 3 **characterized in that** said step of redistribution for each one of the processors comprises an initial phase during which software objects executing on the processor are removed so as to free up memory areas associated with said processor said initial phase being followed by a finishing phase during which software objects to be created on said processor are created.

5. A method of automatically recover from multiple permanent failures in accordance with claim 4 **characterized by** first redistributing software objects that control hardware equipment controlled by the faulty processor and then redistributing software objects not controlling any hardware equipment.

6. A method in accordance with claim 5, **characterized by** blocking software objects that have a hardware equipment dependency and that execute on a faulty processor from access by software objects executing on other operating processors of the processor system.

7. A method in accordance with claim 6, **characterized by** representing a processor by a software object that has a hardware dependency.

8. A method of automatically recover from multiple permanent failures in accordance with claim 5 **characterized in that** said step of creating a catastrophe plan for an individual processor comprises mapping software objects executing on the individual processor on at least two other processors.

9. A method of automatically recover from multiple permanent failures in accordance with claim 6 **characterized by** storing said initial catastrophe plans in a data base distributed in random access memories associated with individual processors of said distributed processor system.

10. A method of automatically recover from multiple permanent failures in accordance with claim 7 **characterized in that** said

catastrophe plans and said new catastrophe plans are created using a first algorithm comprising parameters pertaining to the free capacity and free memory of a processor to take over software objects, parameters pertaining to how much capacity and memory the individual software objects to be redistributed to an individual processor require and parameters relating to the quality of service of the telecommunication system.

11. A method of automatically recover from multiple permanent failures in accordance with claim 8 **characterized in that** said step of returning to the initial configuration implies
- acquisition of information as to which software objects should be brought back to the repaired and installed processor, said acquisition being made with a second algorithm, different from the first,
- killing said software objects to be brought back and
- creating and starting them on the repaired and installed processor.

12. A method of automatically recover from multiple permanent failures in accordance with claim 9 **characterized in that** said second algorithm is implemented by creating a delta configuration, said delta configuration excluding the faulty processor.

13. A method of automatically recover from multiple permanent failures in accordance with claim 1 **characterized in that** the catastrophe plan to be followed if a processor goes down is created by the very same processor.

14. A method of automatically recover from multiple permanent failures of processors in a distributed processor system of a software driven telecommunication system **characterized by** dividing said software into software modules, referred to as software objects, providing said software modules with characteristics making them movable between processors within the processor system, naming said software modules by assigning them individual identities, dividing said processor system into individual processors, naming each processor by assigning them individual identities, and installing the software modules on different ones

of said processors so as to create an initial configuration.

15. A method of automatically recover from multiple permanent failures in accordance with claim 14 **characterized in that** each software module is provided with at least two functions, one that creates the software object and the other that kills the software object.

Fig.1

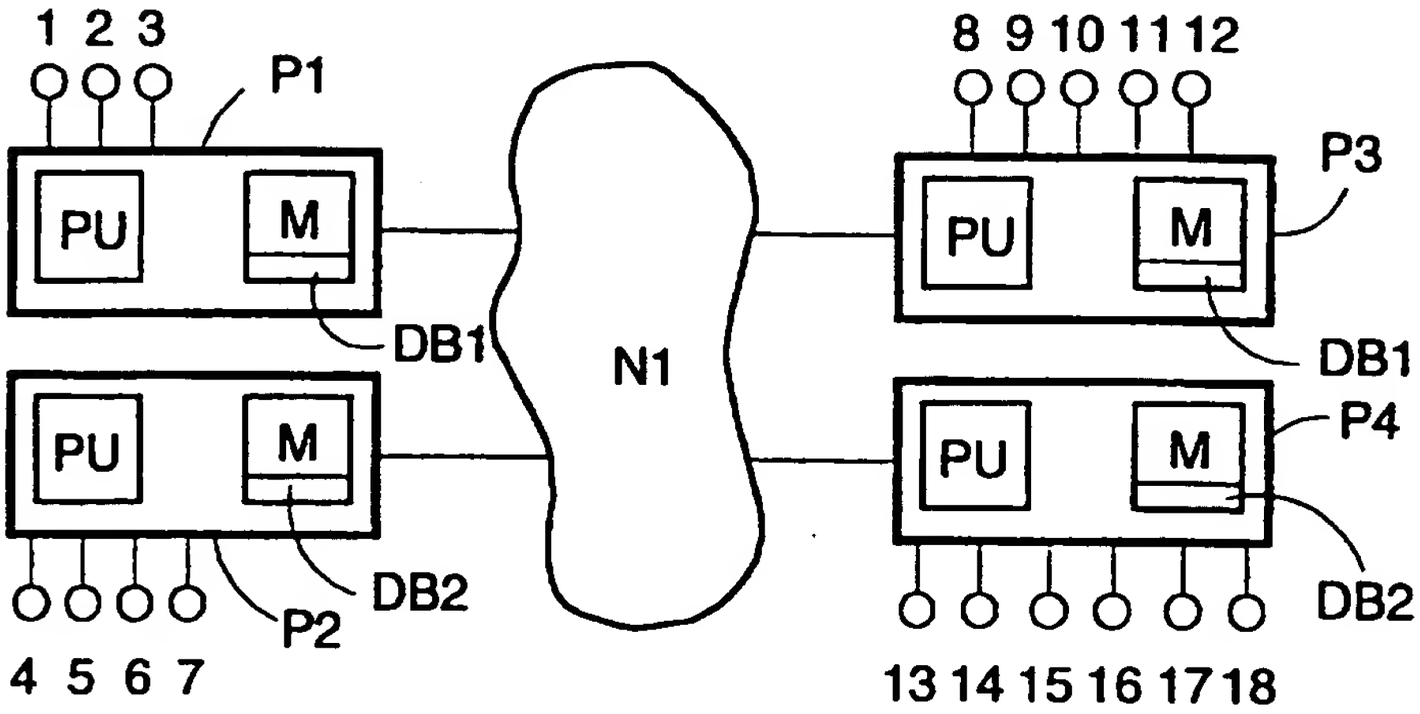


Fig.2

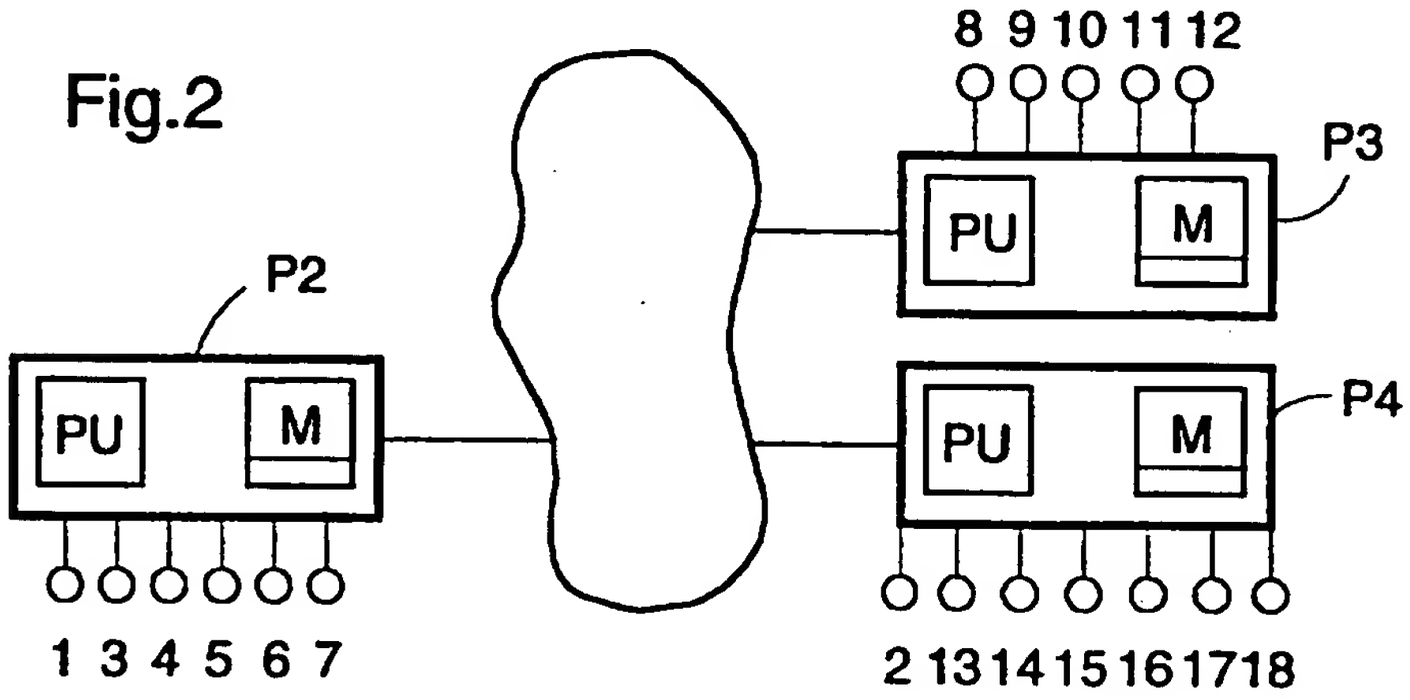


Fig.3

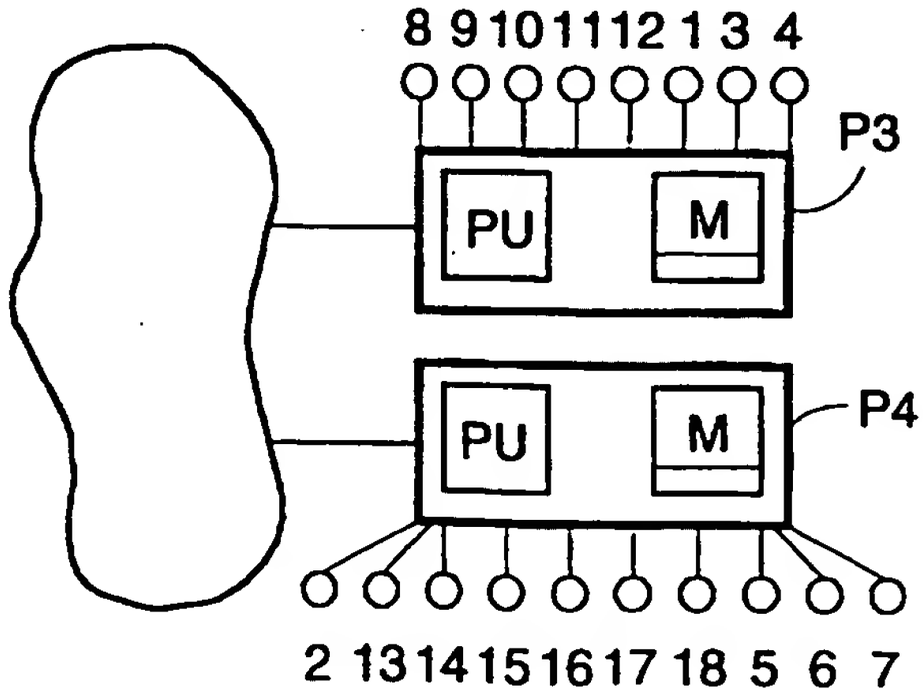


Fig.4

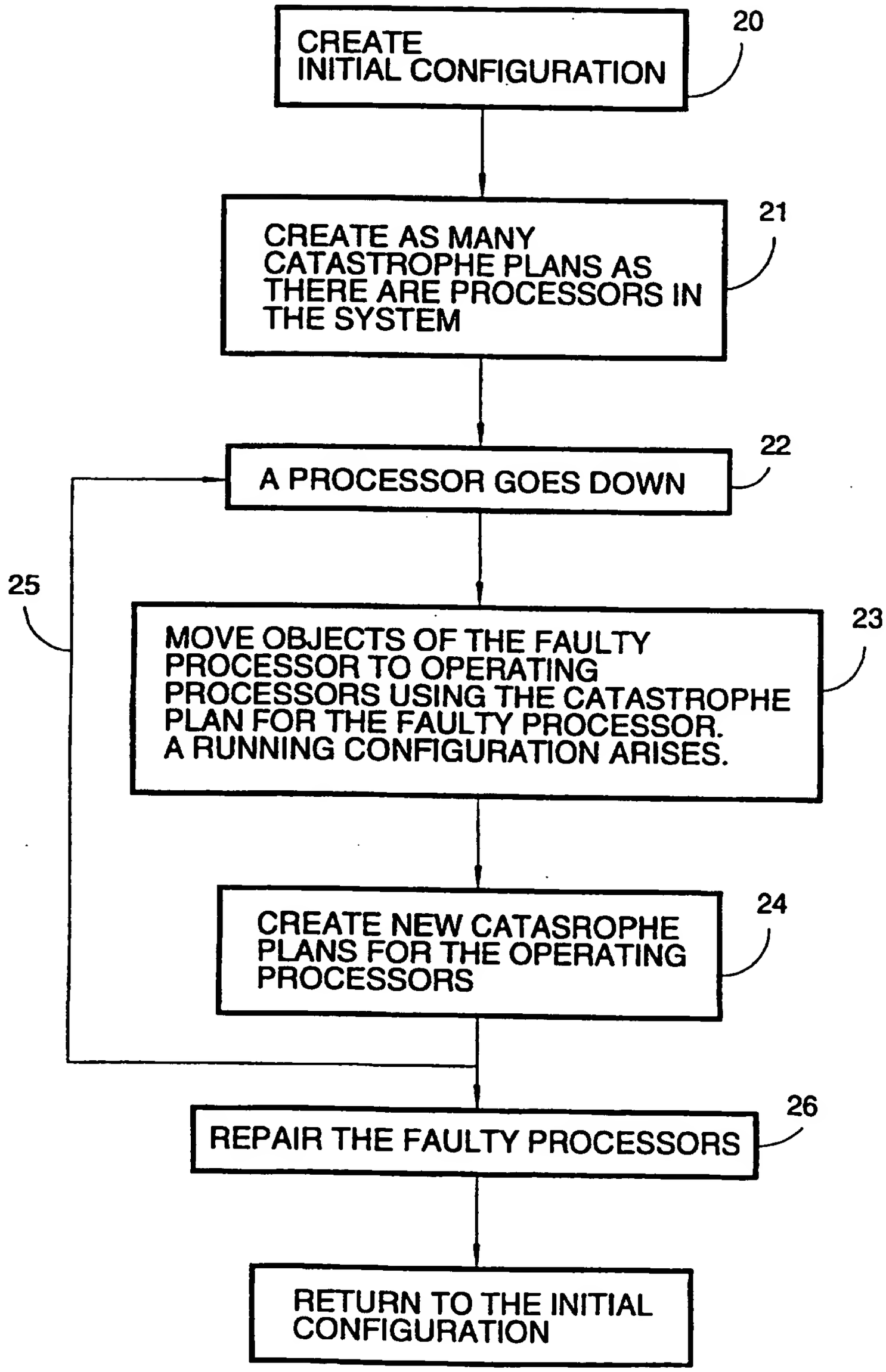

CREATE INITIAL CONFIGURATION
20
CREATE AS MANY CATASTROPHE PLANS AS THERE ARE PROCESSORS IN THE SYSTEM
21
A PROCESSOR GOES DOWN
22
25
MOVE OBJECTS OF THE FAULTY PROCESSOR TO OPERATING PROCESSORS USING THE CATASTROPHE PLAN FOR THE FAULTY PROCESSOR. A RUNNING CONFIGURATION ARISES.
23
CREATE NEW CATASROPHE PLANS FOR THE OPERATING PROCESSORS
24
REPAIR THE FAULTY PROCESSORS
26
RETURN TO THE INITIAL CONFIGURATION

Fig.5

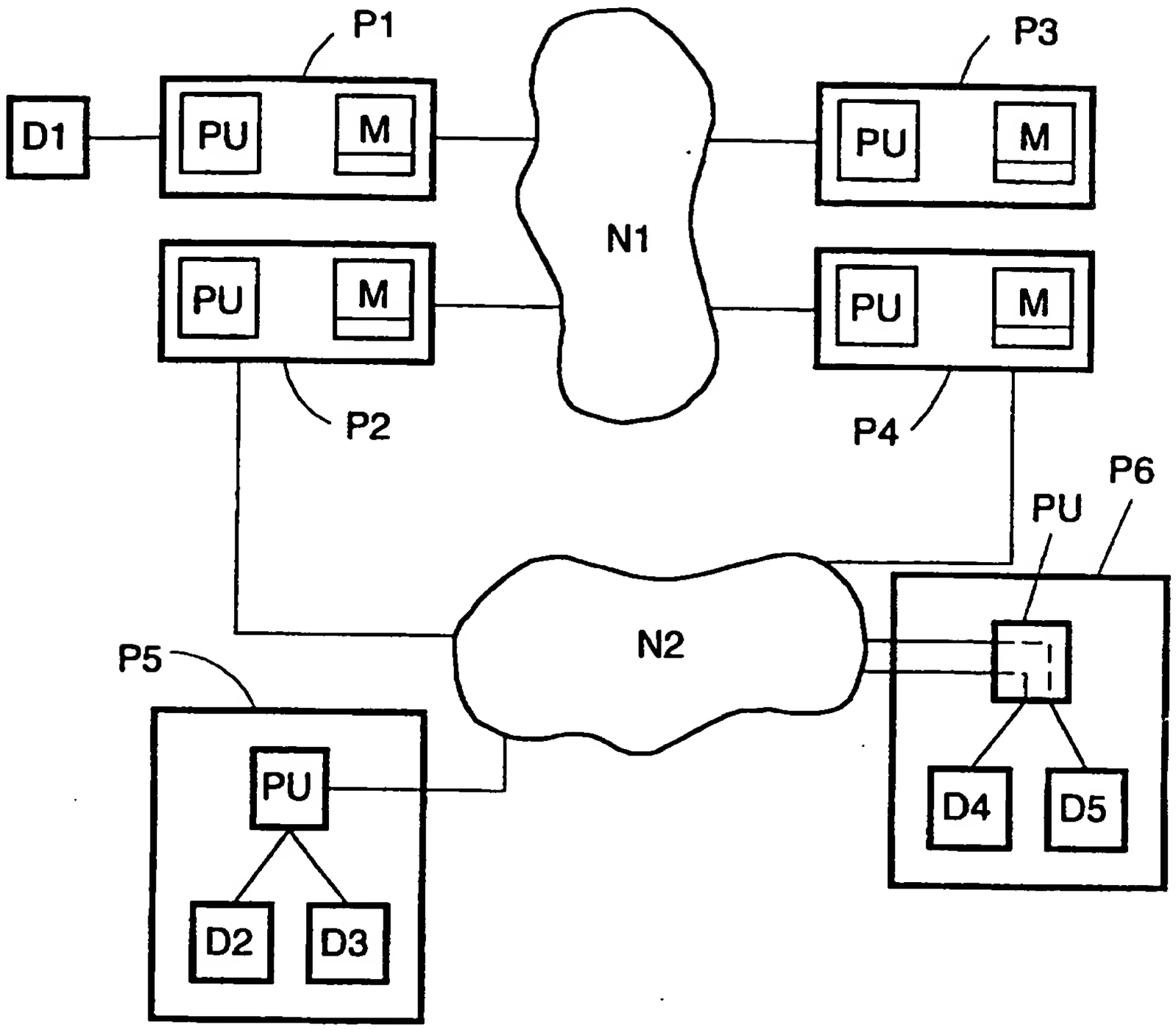
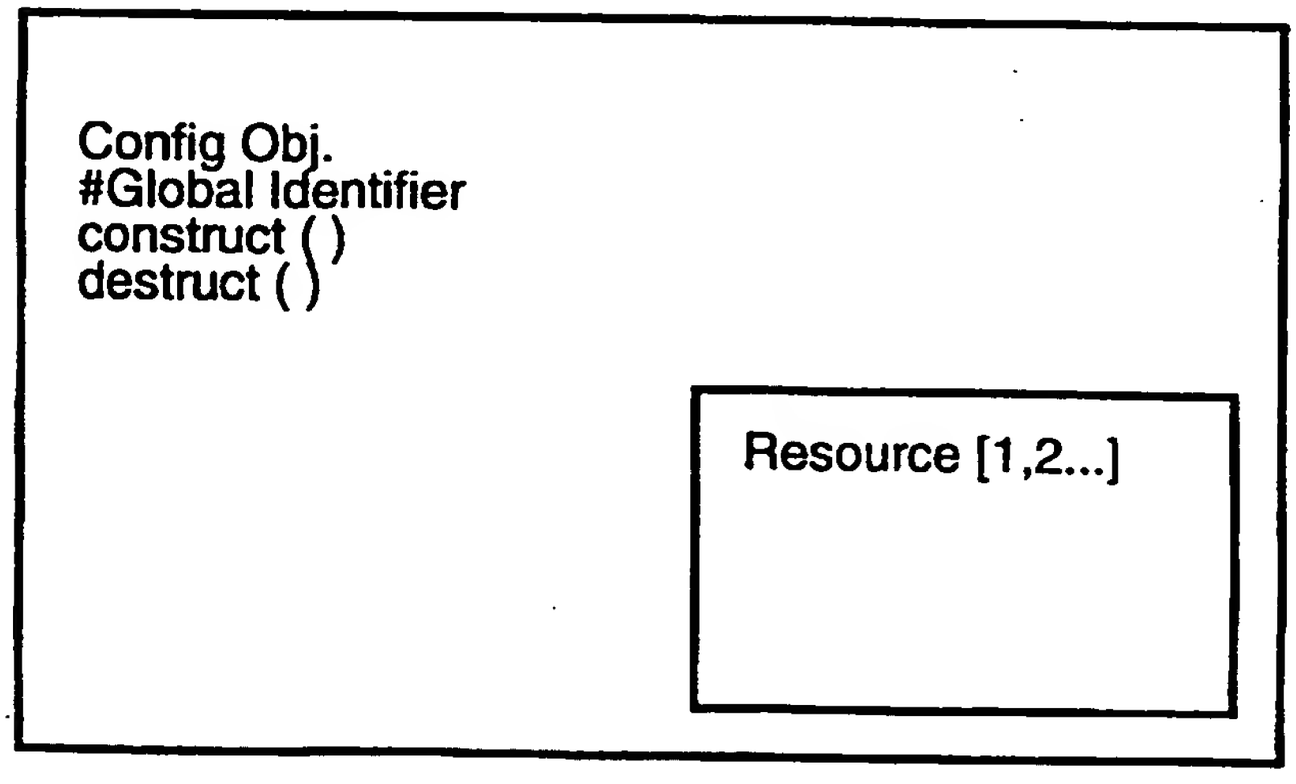

P1
P3
D1
PU
M
PU
M
N1
PU
M
PU
M
P2
P4
P6
PU
N2
P5
PU
D4
D5
D2
D3

Fig.6 A


Config Obj.
#Global Identifier
construct ( )
destruct ( )
Resource [1,2...]

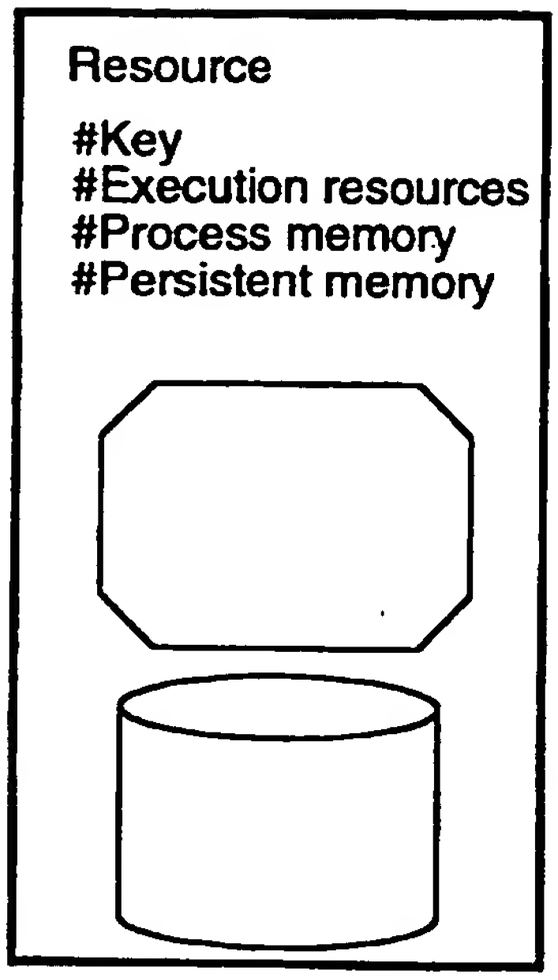


Fig.6B

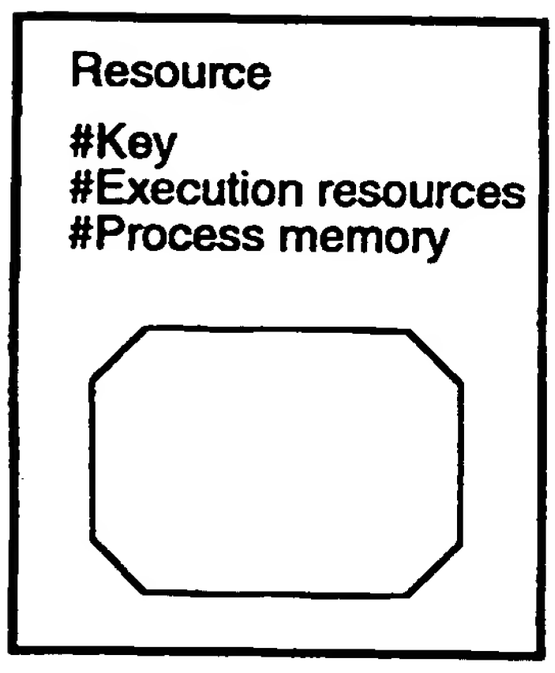


Fig.6C

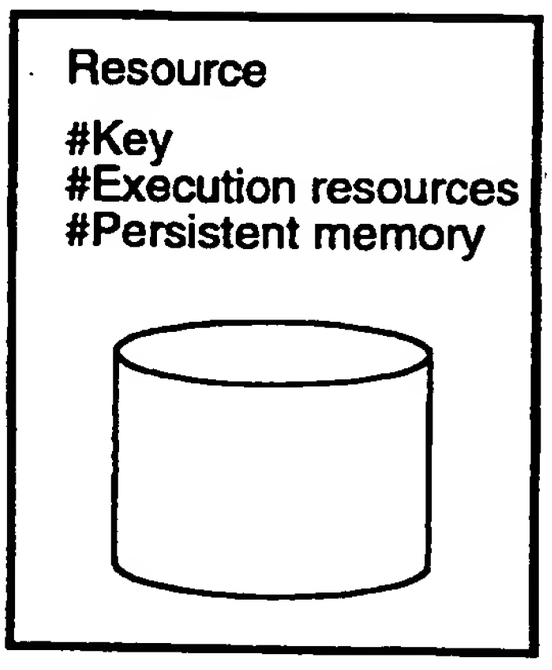


Fig.6D

Fig.7

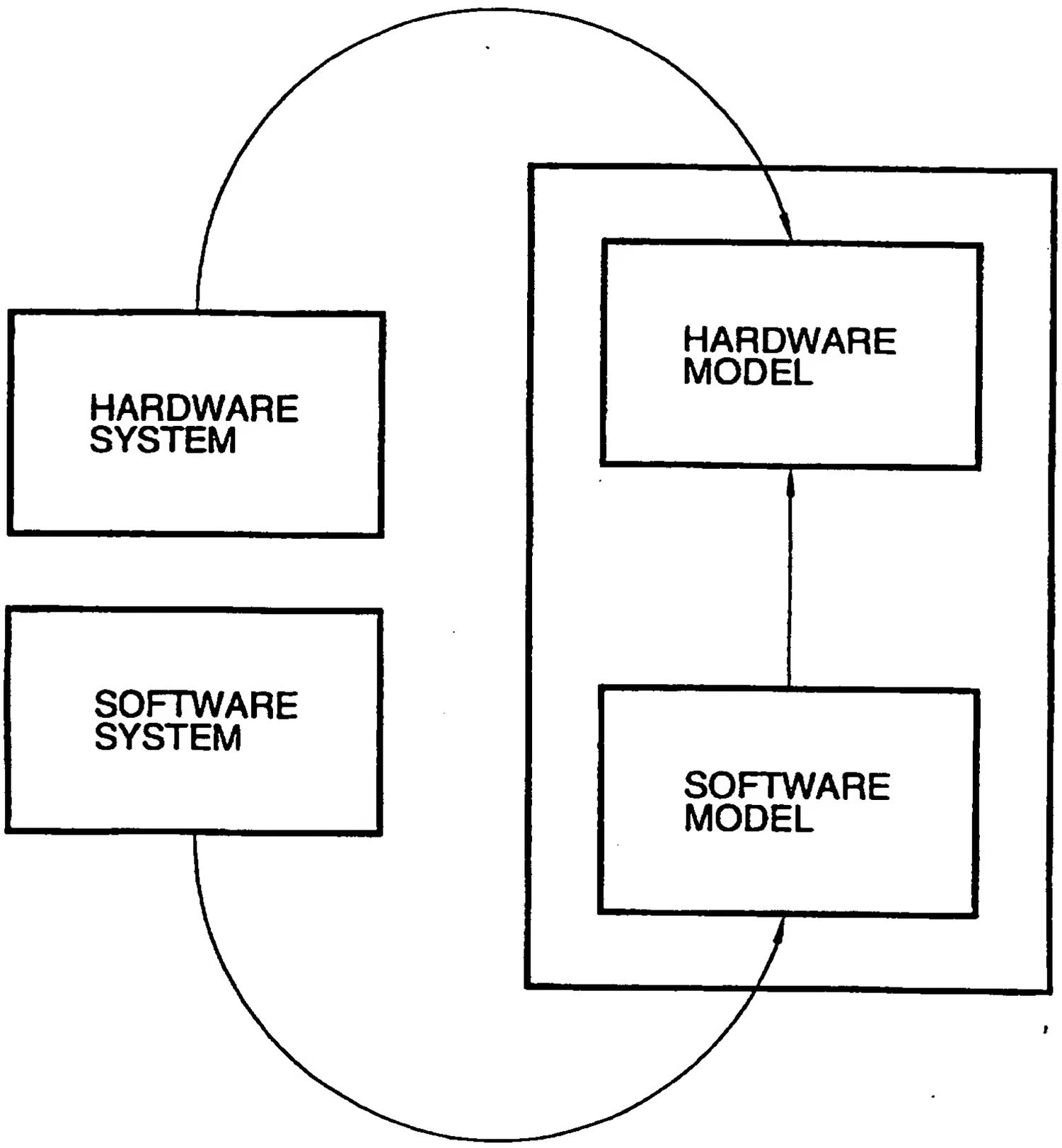

HARDWARE SYSTEM
SOFTWARE SYSTEM
HARDWARE MODEL
SOFTWARE MODEL

Fig.8

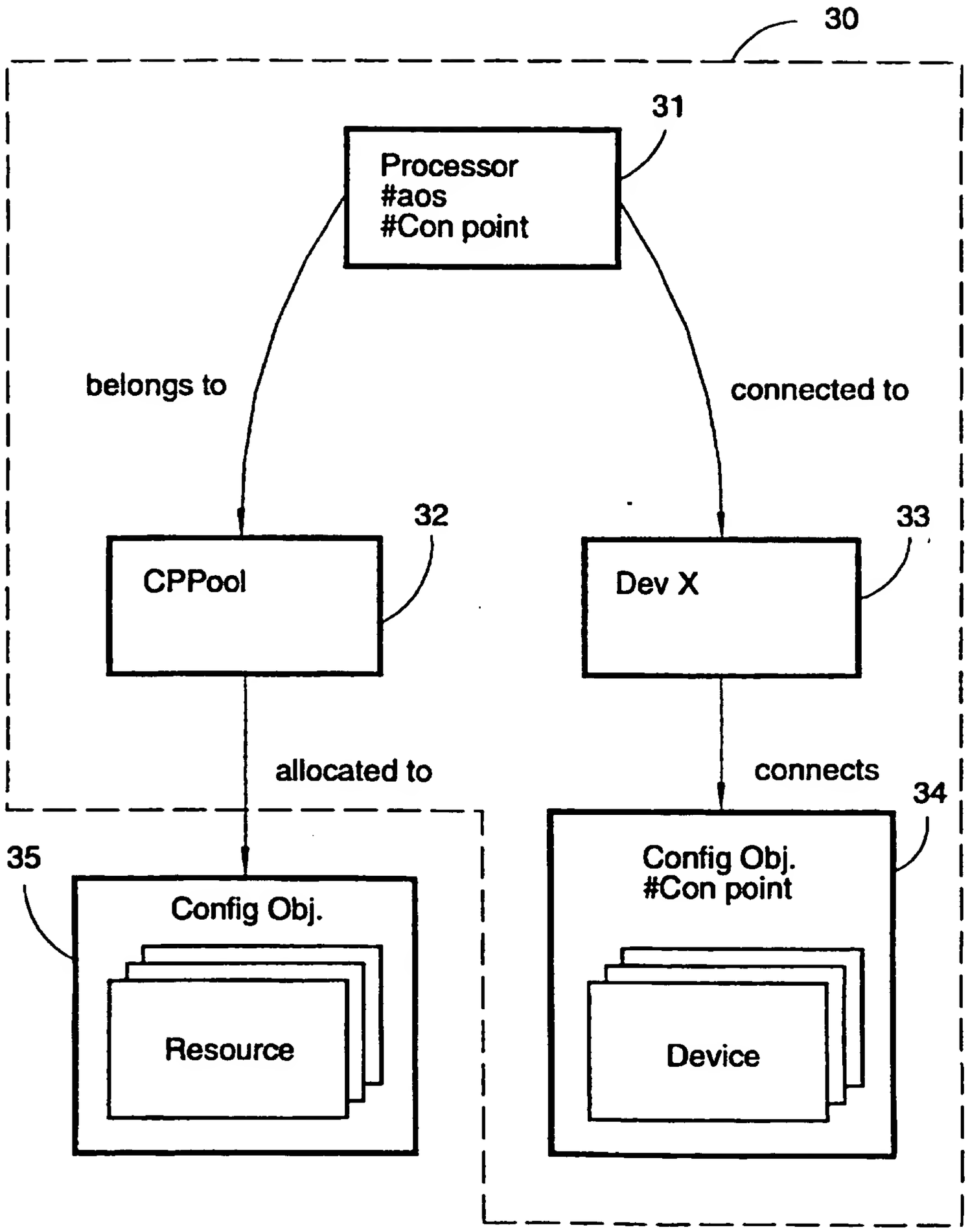

30
31
Processor
#aos
#Con point
belongs to
connected to
32
CPPool
33
Dev X
allocated to
connects
34
35
Config Obj.
Resource
Config Obj.
#Con point
Device

Fig.9